JDK Contents

Reflection

Enables Java code to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions. The API accommodates applications that need access to either the public members of a target object (based on its runtime class) or the members declared by a given class.

Reflection Specification

- **Java Core Reflection Specification**

Reflection API Reference
(javadoc)

- **java.lang.reflect Package**

Reflection FAQ

- **Reflection Frequently Asked Questions**

Reflection Tutorial

- **A New Lesson in the Online Java Tutorial**

Please send comments to: reflection-comments@worthy.eng.sun.com

Overview

The Java(TM) Core Reflection API provides a small, type-safe, and secure API that supports introspection about the classes and objects in the current Java Virtual Machine. If permitted by security policy, the API can be used to:

- construct new class instances and new arrays
- access and modify fields of objects and classes
- invoke methods on objects and classes
- access and modify elements of arrays

The Core Reflection API defines new classes and methods, as follows:

- Three new classes-`Field`, `Method`, and `Constructor`-that reflect class and interface members and constructors. These classes provide:
  - reflective information about the underlying member or constructor
  - a type-safe means to use the member or constructor to operate on Java objects
- New methods of class `Class` that provide for the construction of new instances of the `Field`, `Method`, and `Constructor` classes.
- One new class-`Array`-that provides methods to dynamically construct and access Java arrays.
- One new utility class-`Modifier`-that helps decode Java language modifier information about classes and their members.

There are also some additions to the `java.lang` package that support reflection. These additions are:

- Two new classes-`Byte` and `Short`. These new classes are subclasses of the class `Number`, and are similar to the class `Integer`. Instances of these new classes serve as object wrappers for primitive values of type `byte` and `short`, respectively.
- New objects, instances of the class `Class`, to represent the primitive Java types `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`, and the keyword `void`, at run-time.
- A new, uninstantiable placeholder class-`Void`-to hold a reference to the `Class` object representing the keyword `void`.

Applications

The Core Reflection API accommodates two categories of applications.

One category is comprised of applications that need to discover and use all of the `public` members of a target object based on its run-time class. These applications require run-time access to all the `public` fields, methods, and constructors of an object. Examples in this category are services such as *Java* (TM) Beans[1], and lightweight tools, such as object inspectors. These applications use the instances of the classes `Field`, `Method`, and `Constructor` obtained through the methods `getField`, `getMethod`, `getConstructor`, `getFields`, `getMethods`, and `getConstructors` of class `Class`.

The second category consists of sophisticated applications that need to discover and use the members declared by a given class. These applications need run-time access to the implementation of a class at the level provided by a `class` file. Examples in this category are development tools, such as debuggers, interpreters, inspectors, and class browsers, and run-time services, such as *Java*(TM) Object Serialization[2]. These applications use instances of the classes `Field`, `Method`, and `Constructor` obtained through the methods `getDeclaredField`, `getDeclaredMethod`, `getDeclaredConstructor`, `getDeclaredFields`, `getDeclaredMethods`, and `getDeclaredConstructors` of class `Class`.

Reflection Model

The three classes `Field`, `Method`, and `Constructor` are `final`. Only the Java Virtual Machine may create instances of these classes; these objects are used to manipulate the underlying objects; that is, to:

- get reflective information about the underlying member or constructor
- get and set field values
- invoke methods on objects or classes
- create new instances of classes

The `final` uninstantiable class `Array` provides `static` methods that permit creating new arrays, and getting and setting the elements of arrays.

Member Interface

The classes `Field`, `Method` and `Constructor` implement the `Member` interface. The methods of `Member` are used to query a reflected member for basic identifying information. Identifying information consists of the class or interface that declared the member, the name of the member itself, and the Java language modifiers (such as `public`, `protected`, `abstract`, `synchronized`, and so on) for the member.

Field Objects

A `Field` object represents a reflected field. The underlying field may be a class variable (a `static` field) or an instance variable (a non-static field). Methods of class Field are used to obtain the type of the underlying field, and to get and set the underlying field's value on objects.

Method Objects

A Method object represents a reflected method. The underlying method may be an abstract method, an instance method, or a class (static) method.

Methods of class Method are used to obtain the formal parameter types, the return type, and the checked exception types of the underlying method. In addition, the invoke method of class Method is used to invoke the underlying method on target objects. Instance and abstract method invocation uses dynamic method resolution based on the target object's run-time class and the reflected method's declaring class, name, and formal parameter types. (Thus, it is permissible to invoke a reflected interface method on an object that is an instance of a class that implements the interface.) Static method invocation uses the underlying static method of the method's declaring class.

Constructor Objects

A Constructor object represents a reflected constructor. Methods of class Constructor are used to obtain the formal parameter types and the checked exception types of the underlying constructor. In addition, the newInstance method of class Constructor is used to create and initialize a new instance of the class that declares the constructor, provided the class is instantiable.

Array and Modifier Classes

The Array class is an uninstantiable class that exports class methods to create Java arrays with primitive or class component types. Methods of class Array are also used to get and set array component values.

The Modifier class is an uninstantiable class that exports class methods to decode Java language modifiers for classes and members. The language modifiers are encoded in an integer, and use the encoding constants defined by *The Java Virtual Machine Specification*.

Representation of Primitive Java Types

Finally, there are nine new Class objects that are used to represent the eight primitive Java types and void at run-time. (Note that these are Class *objects*, not classes.) The Core Reflection API uses these objects to identify the following:

- primitive field types
- primitive method and constructor parameter types
- primitive method return types

The Java Virtual Machine creates these nine Class objects. They have the same names as the types that they represent. The Class objects may only be referenced via the following public final static variables:

```
java.lang.Boolean.TYPE

java.lang.Character.TYPE

java.lang.Byte.TYPE

java.lang.Short.TYPE

java.lang.Integer.TYPE

java.lang.Long.TYPE

java.lang.Float.TYPE

java.lang.Double.TYPE

java.lang.Void.TYPE
```

In particular, these `Class` objects are not accessible via the `forName` method of class `Class`.

Security Model

The Java security manager controls access to the Core Reflection API on a class-by-class basis. There are two levels of checks to enforce security and safety, as follows:

- The new methods of class `Class` that give reflective access to a member or a set of members of a class are the only source for instances of `Field`, `Method`, and `Constructor`. These methods first delegate security checking to the system security manager (if installed), which throws a `SecurityException` should the request for reflective access be denied.

- Once the system security manager grants initial reflective access to a member, any code may query the reflected member for its identifying information. However, standard Java language access control checks-for `protected`, default (package) access, and `private` classes and members-will normally occur when the individual reflected members are used to operate on the underlying members of objects,that is, to get or set field values, to invoke methods, or to create and initialize new objects. Unrestricted access, which overrides standard language access control rules, may be granted to privileged code (such as debugger code)-a future version of this specification will define the interface by which this may be accomplished.

The initial policy decision is centralized in a new method of class `SecurityManager`, the `checkMemberAccess` method

```
void checkMemberAccess(Class,int) throws SecurityException
```

The `Class` parameter identifies the class or interface whose members need to be accessed. The `int` parameter identifies the set of members to be accessed-either `Member.PUBLIC` or `Member.DECLARED`.

If the requested access to the specified set of members of the specified class is denied, the method should throw a `SecurityException`. If the requested access to the set is granted, the method should return.

As stated earlier, standard Java language access control will be enforced when a reflected member from this set is used to operate on an underlying object, that is, when:

- a `Field` is used to get or set a field value
- a `Method` is used to invoke a method
- a `Constructor` is used to create and initialize a new instance of a class

If access is denied at that point, the reflected member will throw an `IllegalAccessException`.

Java Language Policy

The Java language security policy for applications is that any code may gain reflective access to all the members and constructors (including non-`public` members and constructors) of any class it may link against. Application code that gains reflective access to a member or constructor may only *use* the reflected member or constructor with standard Java language access control.

JDK 1.1 Security Policy

Sun's Java Development Kit 1.1 (JDK1.1) implements its own security policy that is *not* part of the language specification. In Sun's JDK1.1, the class `AppletSecurity` implements the following policy:

- Untrusted (applet) code is granted access to:
    - All `public` members of all `public` classes loaded by the same class loader as the untrusted code
    - All `public` members of `public` system classes
    - All declared (including non-`public`) members of all classes loaded by the same class loader as the untrusted code
- Trusted (applet) code, defined as code signed by a trusted entity, is additionally granted access to all members of system classes.
- System code, defined as code loaded from `CLASSPATH`, is additionally granted access to all classes loaded by all class loaders.

Any code that gains reflective access to a member may only use it with standard Java language access control. There is no notion of privileged code, and no means to override the standard language access control checks.

This policy is conservative with respect to untrusted code-it is more restrictive than the linker for the Java Virtual Machine. For example, an untrusted class cannot, by itself, access a `protected` member of a system superclass via reflection, although it can via the linker. (However, system code may access such members and pass them to untrusted code.)

The JDK security policy is expected to evolve with the security framework for Java.

Data Conversions

Certain methods in the reflection package perform automatic data conversions between values of primitive types and objects of class types. These are the generic methods for getting and setting field and array component values, and the methods for method and constructor invocation.

There are two types of automatic data conversions. *Wrapping conversions* convert from values of primitive types to objects of class types. *Unwrapping conversions* convert objects of class types to values of primitive types. The rules for these conversions are defined in "Wrapping and Unwrapping Conversions."

Additionally, field access and method invocation permit *widening conversions* on primitive and reference types. These conversions are documented in *The Java Language Specification*, section 5, and are detailed in "Widening Conversions."

Wrapping and Unwrapping Conversions

A primitive value is automatically wrapped in an object when it is retrieved via `Field.get` or `Array.get`, or when it is returned by a method invoked via `Method.invoke`.

Similarly, an object value is automatically unwrapped when supplied as a parameter in a context that requires a value of a primitive type. These contexts are:

- `Field.set`, where the underlying field has a primitive type
- `Array.set`, where the underlying array has a primitive element type
- `Method.invoke` or `Constructor.newInstance`, where the corresponding formal parameter of the underlying method or constructor has a primitive type

The following table shows the correspondences between primitive types and class (wrapper) types:

`boolean`	`java.lang.Boolean`
`char`	`java.lang.Character`
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`

A method that is declared `void` returns the special reference `null` when it is invoked via `Method.invoke`.

Widening Conversions

The reflection package permits the same widening conversions at run-time as permitted in method invocation contexts at compile time. These conversions are defined in *The Java Language Specification*, section 5.3.

Widening conversions are performed at run-time:

- when a value is retrieved from a field or an array via the methods of `Field` and `Array`
- when a value is stored into a field or an array via the methods of `Field` and `Array`
- when an unwrapped actual parameter value is converted to the type of its corresponding formal parameter during method or constructor invocation via `Method.invoke` or `Constructor.newInstance`

The permitted *widening primitive conversions* are:

- From `byte` to `short`, `int`, `long`, `float`, or `double`
- From `short` to `int`, `long`, `float`, or `double`
- From `char` to `int`, `long`, `float`, or `double`
- From `int` to `long`, `float`, or `double`
- From `long` to `float` or `double`
- From `float` to `double`.

The permitted *widening reference conversions* are:

- From a class type *S* to a class type *T*, provided that *S* is a subclass of *T*
- From a class type *S* to an interface type *K*, provided that *S* implements *K*
- From an interface type *J* to an interface type *K*, provided that *J* is a subinterface of *K*

Packaging

The Core Reflection API is in a new subpackage of `java.lang` named `java.lang.reflect`. This avoids compatibility problems caused by Java's default package importation rules.